Λειτουργικά Συστήματα Υπολογιστών
7ο Εξάμηνο, ΣΗΜΜΥ ΕΜΠ
22/12/2016

Αναφορά 3ης Άσκησης

Ομάδα Ε15
Γαβαλάς Νικόλαος 03113121
Καραΐσκου Κωνσταντίνα 03113127

Άσκηση 1.1

Source Code:

```
/*
 * simplesync.c
 *
 * A simple synchronization exercise.
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 * Operating Systems course, ECE, NTUA
 *
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

/*
 * POSIX thread functions do not return error numbers in errno,
 * but in the actual return value of the function call instead.
 * This macro helps with error reporting in this case.
 */
#define perror_pthread(ret, msg) \
        do { errno = ret; perror(msg); } while (0)
```

```c
#define N 10000000

/* Dots indicate lines where you are free to insert code at will */
/* ... */
pthread_mutex_t mutex;            // δήλωση του κλειδώματος mutex
int key=0;                  // η μεταβλητή για τα atomic operations

#if defined(SYNC_ATOMIC) ^ defined(SYNC_MUTEX) == 0
# error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
#endif

#if defined(SYNC_ATOMIC)
# define USE_ATOMIC_OPS 1
#else
# define USE_ATOMIC_OPS 0
#endif

void *increase_fn(void *arg)
{
        int i;
        volatile int *ip = arg;

        fprintf(stderr, "About to increase variable %d times\n", N);
        for (i = 0; i < N; i++) {
                if (USE_ATOMIC_OPS) {
                        /* αυξάνουν κατά 1 την τιμή key και επιστρέφουν την αρχική τιμή του
                        key πριν την αύξηση */
                        __sync_fetch_and_add(&key,1);
                        /* You can modify the following line */
                        if(key==0)      // το κρίσιμο τμήμα εκτελείται μόνο για key=0
                                ++(*ip);
                        /* μειώνουν κατά 1 την τιμή key και επιστρέφουν την αρχική τιμή του
                        key πριν την μείωση*/
                        __sync_fetch_and_sub(&key,1);
                } else {
                        /* λήψη του κλειδώματος αμοιβαίου αποκλεισμού mutex */
                        pthread_mutex_lock(&mutex);
                        /* You cannot modify the following line */
                        ++(*ip);
                        /* αποδέσμευση του κλειδώματος αμοιβαίου αποκλεισμού mutex*/
                        pthread_mutex_unlock(&mutex);
```

```
                }
        }
        fprintf(stderr, "Done increasing variable.\n");

        return NULL;
}

void *decrease_fn(void *arg)
{
        int i;
        volatile int *ip = arg;

        fprintf(stderr, "About to decrease variable %d times\n", N);
        for (i = 0; i < N; i++) {
                if (USE_ATOMIC_OPS) {
                        /* αυξάνουν κατά 1 την τιμή key και επιστρέφουν την αρχική τιμή του
                        key πριν την αύξηση */
                        __sync_fetch_and_add(&key,1);
                        /* You can modify the following line */
                        if(key==0)        // το κρίσιμο τμήμα εκτελείται μόνο για key=0
                                --(*ip);
                        /* μειώνουν κατά 1 την τιμή key και επιστρέφουν την αρχική τιμή του
                        key πριν την μείωση*/
                        __sync_fetch_and_sub(&key,1);
                } else {
                        /* λήψη του κλειδώματος αμοιβαίου αποκλεισμού mutex */
                        pthread_mutex_lock(&mutex);
                        /* You cannot modify the following line */
                        --(*ip);
                        /* αποδέσμευση του κλειδώματος αμοιβαίου αποκλεισμού mutex */
                        pthread_mutex_unlock(&mutex);
                }
        }
        fprintf(stderr, "Done decreasing variable.\n");

        return NULL;
}
```

```c
int main(int argc, char *argv[])
{
        int val, ret, ok;
        pthread_t t1, t2;

        /*
        * Initial value
        */
        val = 0;

        /*
        * Create threads
        */
        ret = pthread_create(&t1, NULL, increase_fn, &val);
        if (ret) {
                perror_pthread(ret, "pthread_create");
                exit(1);
        }
        ret = pthread_create(&t2, NULL, decrease_fn, &val);
        if (ret) {
                perror_pthread(ret, "pthread_create");
                exit(1);
        }

        /*
        * Wait for threads to terminate
        */
        ret = pthread_join(t1, NULL);
        if (ret)
                perror_pthread(ret, "pthread_join");
        ret = pthread_join(t2, NULL);
        if (ret)
                perror_pthread(ret, "pthread_join");

        /*
        * Is everything OK?
        */
        ok = (val == 0);
        printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);
        return ok;
}
```

Παραδείγματα Εκτέλεσης:

```
oslabe15@kythnos:~/erg3$ ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
```

```
oslabe15@kythnos:~/erg3$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Απαντήσεις στις ερωτήσεις:

1. Χρησιμοποιήστε την εντολή time(1) για να μετρήσετε το χρόνο εκτέλεσης των εκτελέσιμων. Πώς συγκρίνεται ο χρόνος εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό, σε σχέση με το χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό; Γιατί;

Ο χρόνος εκτέλεσης του αρχικού προγράμματος είναι
```
real    0m0.079s
user    0m0.140s
sys     0m0.000s,
```

που είναι πολύ μικρότερος από το χρόνο εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό. Αυτό συμβαίνει διότι στο αρχικό πρόγραμμα τα νήματα δεν περιμένουν τη λήψη κλειδώματος για να εκτελέσουν την κρίσιμη περιοχή κι ύστερα να αποδεσμεύσουν το κλείδωμα.

2. Ποια μέθοδος συγχρονισμού είναι γρηγορότερη, η χρήση ατομικών λειτουργιών ή η χρήση POSIX mutexes; Γιατί;

Χρησιμοποιώντας την εντολή time, βλέπουμε ότι ο χρόνος εκτέλεσης του simplesync-mutex, που χρησιμοποιεί POSIX mutexes, είναι:
```
real    0m6.101s
user    0m6.484s
```

```
sys     0m5.672s,
```

ενώ ο χρόνος εκτέλεσης του simplesync-atomic που χρησιμοποιεί ατομικές λειτουργίες είναι:
real 0m2.702s
user 0m5.384s
sys 0m0.000s.

Διαπιστώνουμε ότι η χρήση ατομικών λειτουργιών είναι γρηγορότερη μέθοδος συγχρονισμού, όπως αναμέναμε, καθώς τα mutexes χρησιμοποιούν τα atomic operations για την λήψη και την αποδέσμευση ενός mutex. Επιπλέον, τα atomic operations υλοποιούνται σε επίπεδο hardware, και άρα είναι γρηγορότερα από τα mutexes που είναι επιπέδου software.

3. Σε ποιες εντολές του επεξεργαστή μεταφράζεται η χρήση ατομικών λειτουργιών του GCC στην αρχιτεκτονική για την οποία μεταγλωττίζετε; Χρησιμοποιήστε την παράμετρο -S του GCC για να παράγετε τον ενδιάμεσο κώδικα Assembly, μαζί με την παράμετρο -g για να συμπεριλάβετε πληροφορίες γραμμών πηγαίου κώδικα (π.χ., ".loc 1 63 0"), οι οποίες μπορεί να σας διευκολύνουν Δείτε την έξοδο της εντολής make για τον τρόπο μεταγλώττισης του simplesync.c.

Η χρήση ατομικών λειτουργιών στην αρχιτεκτονική την οποία μεταγλωττίζουμε μεταφράζεται στις ακόλουθες εντολές:

- για την `__sync_fetch_and_add(&key,1):`

```
.loc 1 51 0
lock addl       $1, key
```

- για την `__sync_fetch_and_sub(&key,1):`

```
.loc 1 56 0
lock subl       $1, key
```

4. Σε ποιες εντολές μεταφράζεται η χρήση POSIX mutexes στην αρχιτεκτονική για την οποία μεταγλωττίζετε; Παραθέστε παράδειγμα μεταγλώττισης λειτουργίας pthread_mutex_lock() σε Assembly, όπως στο προηγούμενο ερώτημα.

Η χρήση POSIX mutexes στην αρχιτεκτονική για την οποία μεταγλωττίζουμε μεταφράζεται στις ακόλουθες εντολές:

- για την `void *increase_fn(void *arg):`

```
.cfi_def_cfa_offset 32
call    pthread_mutex_lock
```

```
.LVL4:
        .loc 1 61 0
        movl    (%esi), %eax
        addl    $1, %eax
        movl    %eax, (%esi)
        .loc 1 63 0
        movl    $mutex, (%esp)
        call    pthread_mutex_unlock
```

- για την `void *decrease_fn(void *arg):`

```
        .cfi_def_cfa_offset 32
        call    pthread_mutex_lock
.LVL14:
        .loc 1 90 0
        movl    (%esi), %eax
        subl    $1, %eax
        movl    %eax, (%esi)
        .loc 1 92 0
        movl    $mutex, (%esp)
        call    pthread_mutex_unlock
```

Άσκηση 1.2

Source Code:

```c
/*
  Source Code:
 *
 * A program to draw the Mandelbrot Set on a 256-color xterm.
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
```

```c
#include <pthread.h>
#include <errno.h>
#include <semaphore.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

#define perror_pthread(ret,msg) \
        do { errno = ret; perror(msg); } while(0)

/***************************
 * Compile-time parameters *
 ***************************/

int NTHREADS;

/*
 * Output at the terminal is is x_chars wide by y_chars long
*/
int y_chars = 50;
int x_chars = 90;

/*
 * The part of the complex plane to be drawn:
 * upper left corner is (xmin, ymax), lower right corner is (xmax, ymin)
*/
double xmin = -1.8, xmax = 1.0;
double ymin = -1.0, ymax = 1.0;

/*
 * Every character in the final output is
 * xstep x ystep units wide on the complex plane.
 */
double xstep;
double ystep;

// struct με πληροφορίες για κάθε νήμα
struct thread_info_struct {
        pthread_t tid;
```

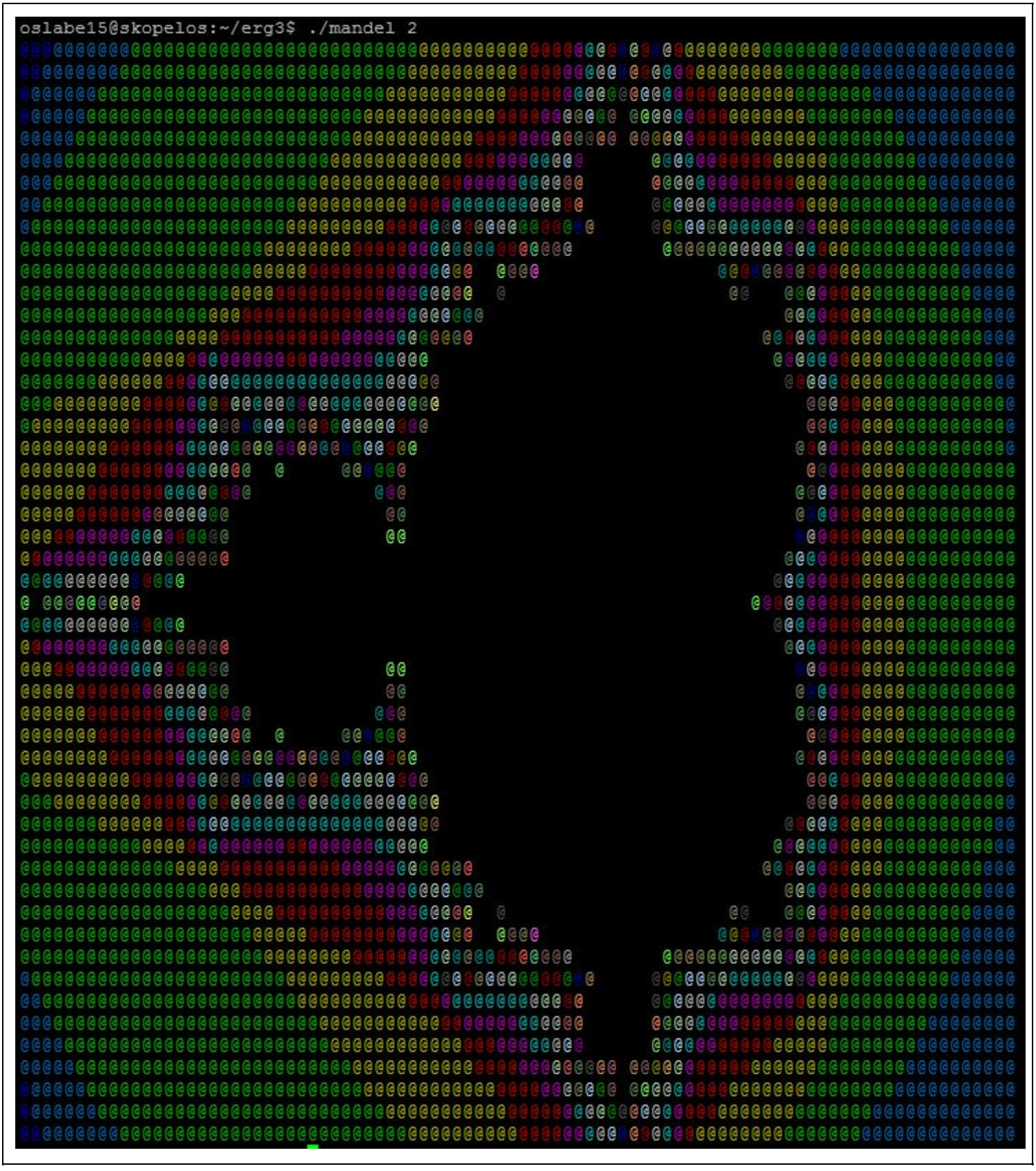
```
        sem_t sem;
};

struct thread_info_struct *thr;

/*
 * This function computes a line of output
 * as an array of x_char color values.
 */
void compute_mandel_line(int line, int color_val[])
{
        /*
        * x and y traverse the complex plane.
        */
        double x, y;

        int n;
        int val;

        /* Find out the y value corresponding to this line */
        y = ymax - ystep * line;

        /* and iterate for all points on this line */
        for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {

                /* Compute the point's color value */
                val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
                if (val > 255)
                        val = 255;

                /* And store it in the color_val[] array */
                val = xterm_color(val);
                color_val[n] = val;
        }
}

/*
 * This function outputs an array of x_char color values
 * to a 256-color xterm.
 */
void output_mandel_line(int fd, int color_val[])
```

```
{
        int i;

        char point ='@';
        char newline='\n';

        for (i = 0; i < x_chars; i++) {
                /* Set the current color, then output the point */
                set_xterm_color(fd, color_val[i]);
                if (write(fd, &point, 1) != 1) {
                        perror("compute_and_output_mandel_line: write point");
                        exit(1);
                }
        }

        /* Now that the line is done, output a newline character */
        if (write(fd, &newline, 1) != 1) {
                perror("compute_and_output_mandel_line: write newline");
                exit(1);
        }
}

void compute_and_output_mandel_line(int fd , int line)
{
        /*
         * A temporary array, used to hold color values for the line being drawn
         */
        int color_val[x_chars];

        compute_mandel_line(line, color_val);
        output_mandel_line(fd, color_val);
}

void *thread_fn(void *i)
{
        int color_val[x_chars];
        int fd=1;
        int line,j=(int)i;

        // κάθε j νήμα αναλαμβάνει τις γραμμές j, j+NTHREADS, j+2*NTHREADS...
        for (line=j; line<y_chars; line=line+NTHREADS)
```

```
        {
                compute_mandel_line(line, color_val);
                // λήψη σημαφόρου
                sem_wait(&thr[j].sem);

                output_mandel_line(fd, color_val);
                // αποδέσμευσε το σημαφόρο που αντιστοιχεί στην επόμενη γραμμή
                if ((j+1)<NTHREADS)
                {
                        sem_post(&thr[j+1].sem);
                }
                else {
                        sem_post(&thr[0].sem);
                 }
        }
        return NULL;
}

int main(int argc, char *argv[])
{
        int ret, i;

        if(argc!=2)
        {
                printf("Exactly one argument required: the number of threads to
create.\n");
                exit(1);
        }

        NTHREADS= atoi(argv[1]);

        // πρέπει να ισχύει 0<NTHREADS<51
        if(NTHREADS<1 || NTHREADS>50)
        {
                fprintf(stderr,"%d must be less than 50 and over 0 \n",NTHREADS);
                exit(1);
        }

        thr = malloc(NTHREADS * sizeof(struct thread_info_struct));

        // δημιουργία του σημαφόρου που αντιστοιχεί στο πρώτο νήμα και αρχικοποίηση του σε 1
```

```
        if (sem_init(&thr[0].sem,0,1)== -1)
                exit(1);
        // δημιουργία των υπόλοιπων σημαφόρων και αρχικοποίηση τους στο 0
        for (i=1; i<NTHREADS; i++) {
        if (sem_init(&thr[i].sem,0,0)== -1)
                exit(1);
        }

        xstep = (xmax - xmin) / x_chars;
        ystep = (ymax - ymin) / y_chars;

        // δημιούργησε τα νήματα, με όρισμα για τη συναρτηση τον αριθμό του νήματος κάθε φορά
        for (i=0; i<NTHREADS; i++)
        {
                ret= pthread_create (&thr[i].tid, NULL, thread_fn,(void *)i);
                if(ret)
                {
                        perror_pthread(ret,"pthread_create");
                        exit(1);
                }
        }

        // αναμονή για τον τερματισμό των νημάτων
        for (i=0; i<NTHREADS; i++)
        {
                ret= pthread_join (thr[i].tid, NULL);
                if (ret)
                {
                        perror_pthread(ret,"pthread_join");
                        exit(1);
                }
        }


        reset_xterm_color(1);
        return 0;
}
```

Παράδειγμα Εκτέλεσης:



Απαντήσεις στις ερωτήσεις:

1. Πόσοι σημαφόροι χρειάζονται για το σχήμα συγχρονισμού που υλοποιείτε;

Για το σχήμα συγχρονισμού που υλοποιήσαμε απαιτούνται Ν σημαφόροι, όσα και τα νήματα. Καθώς θέλουμε να τυπώνονται οι γραμμές με τη σειρά, πρέπει κάθε φορά να είναι αρχικοποιημένος στο 1 ο σημαφόρος που αντιστοιχεί στο νήμα που τυπώνει τη γραμμή που έχει σειρά. Άρα, με κυκλικό τρόπο, αφότου τυπωθεί η γραμμή που πρέπει, αποδεσμεύουμε το σημαφόρο που αντιστοιχεί στο επόμενο νήμα.

2. Πόσος χρόνος απαιτείται για την ολοκλήρωση του σειριακού και του παράλληλου προγράμματος με δύο νήματα υπολογισμού; Χρησιμοποιήστε την εντολή time(1) για να χρονομετρήσετε την εκτέλεση ενός προγράμματος, π.χ., time sleep 2. Για να έχει νόημα η μέτρηση, δοκιμάστε σε ένα μηχάνημα που διαθέτει επεξεργαστή δύο πυρήνων. Χρησιμοποιήστε την εντολή cat /proc/cpuinfo για να δείτε πόσους υπολογιστικούς πυρήνες διαθέτει κάποιο μηχάνημα.

Για την ολοκλήρωση του σειριακού προγράμματος έχουμε:

```
real    0m1.258s
user    0m1.196s
sys     0m0.032s,
```

ενώ για την ολοκλήρωση του παράλληλου προγράμματος με δύο νήματα υπολογισμού έχουμε:

```
real    0m0.761s
user    0m1.196s
sys     0m0.040s.
```

Τις μετρήσεις τις πήραμε σε μηχάνημα που διαθέτει δύο υπολογιστικούς πυρήνες.

3. Το παράλληλο πρόγραμμα που φτιάξατε, εμφανίζει επιτάχυνση; Αν όχι, γιατί; Τι πρόβλημα υπάρχει στο σχήμα συγχρονισμού που έχετε υλοποιήσει; Υπόδειξη: Πόσο μεγάλο είναι το κρίσιμο τμήμα; Χρειάζεται να περιέχει και τη φάση υπολογισμού και τη φάση εξόδου κάθε γραμμής που παράγεται;

Το παράλληλο πρόγραμμα που φτιάξαμε εμφανίζει επιτάχυνση καθώς τα νήματα εκτελούν ταυτόχρονα τμήματα κώδικα. Για παράδειγμα, μπορεί το ένα νήμα να υπολογίζει μία γραμμή ενώ το άλλο να τυπώνει στην έξοδο τη δική του. Αυτό συμβαίνει διότι το μηχάνημα είχε δύο υπολογιστικούς πυρήνες. Σε περίπτωση που έχουμε έναν υπολογιστικό πυρήνα έχουμε έναν PC που “μεταπηδά” από τον κώδικα

του ενός νήματος στον κώδικα άλλου με αποτέλεσμα να μην υπάρχει επιτάχυνση στο πρόγραμμα.

4. Τι συμβαίνει στο τερματικό αν πατήσετε Ctrl-C ενώ το πρόγραμμα εκτελείται; σε τι κατάσταση αφήνεται, όσον αφορά το χρώμα των γραμμάτων; Πώς θα μπορούσατε να επεκτείνετε το mandel.c σας ώστε να εξασφαλίσετε ότι ακόμη κι αν ο χρήστης πατήσει Ctrl-C, το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του;

Όταν πατήσουμε Ctrl-C ενώ το πρόγραμμα εκτελείται, αυτό τερματίζει και παραμένει στο τερματικό το χρώμα από τον τελευταίο χαρακτήρα που τυπώθηκε, όπως συμβαίνει παρακάτω:

Αυτό συμβαίνει διότι η reset_xterm_color(1), που επαναφέρει το τερματικό στην προηγούμενη κατάστασή του, εκτελείται στο τέλος του προγράμματος. Για να μη συμβαίνει αυτό, μπορούμε να προσθέσουμε στο πρόγραμμα έναν signal handler ώστε μόλις στείλουμε SIGINT με Ctrl-C να το κάνει handle καλωντας την reset_xterm_color(1) πριν τερματίσει.

Άσκηση 1.3

Source Code:

```
/* kindergarten simulator.
 * Bad things happen if teachers and children
 * are not synchronized properly.
 *
 *
 * Author:
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 *
 * Additional Authors:
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 * Anastassios Nanos <ananos@cslab.ece.ntua.gr>
 * Operating Systems course, ECE, NTUA
 *
 */

#include <time.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
/*
 * POSIX thread functions do not return error numbers in errno,
 * but in the actual return value of the function call instead.
 * This macro helps with error reporting in this case.
 */
#define perror_pthread(ret, msg) \
        do { errno = ret; perror(msg); } while (0)

/* A virtual kindergarten */
struct kgarten_struct {

        /*
          * Here you may define any mutexes / condition variables / other variables
          * you may need.
          */
```

```
        pthread_mutex_t lock;                   //mutex
        pthread_cond_t can_enter;               //cond variable for enter
        pthread_cond_t can_exit;                //cond variable for exit

        /*
         * You may NOT modify anything in the structure below this
         * point.
         */
        int vt;
        int vc;
        int ratio;

        pthread_mutex_t mutex;
};

/*
 * A (distinct) instance of this structure
 * is passed to each thread
 */
struct thread_info_struct {
        pthread_t tid; /* POSIX thread id, as returned by the library */

        struct kgarten_struct *kg;
        int is_child;    /* Nonzero if this thread simulates children, zero otherwise */

        int thrid;           /* Application-defined thread id */
        int thrcnt;
        unsigned int rseed;
};

int safe_atoi(char *s, int *val)
{
        long l;
        char *endp;

        l = strtol(s, &endp, 10);
        if (s != endp && *endp == '\0') {
                *val = l;
                return 0;
        } else
                return -1;
```

```
}

void *safe_malloc(size_t size)
{
        void *p;

        if ((p = malloc(size)) == NULL) {
                fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
                        size);
                exit(1);
        }

        return p;
}

void usage(char *argv0)
{
        fprintf(stderr, "Usage: %s thread_count child_threads c_t_ratio\n\n"
                "Exactly two arguments required:\n"
                "    thread_count: Total number of threads to create.\n"
                "    child_threads: The number of threads simulating children.\n"
                "    c_t_ratio: The allowed ratio of children to teachers.\n\n",
                argv0);
        exit(1);
}

void bad_thing(int thrid, int children, int teachers)
{
        int thing, sex;
        int namecnt, nameidx;
        char *name, *p;
        char buf[1024];

        char *things[] = {
                "Little %s put %s finger in the wall outlet and got electrocuted!",
                "Little %s fell off the slide and broke %s head!",
                "Little %s was playing with matches and lit %s hair on fire!",
                "Little %s drank a bottle of acid with %s lunch!",
                "Little %s caught %s hand in the paper shredder!",
                "Little %s wrestled with a stray dog and it bit %s finger off!"
        };
```

```
        char *boys[] = {
                "Makis", "Sakis", "Nick", "Lakis", "Takis",
                "Chris", "Peter", "Paul", "Steve", "Billy", "Mike",
                "Vangelis", "Antony"
        };
        char *girls[] = {
                "Maria", "Irene", "Christina", "Helena", "Georgia", "Olga",
                "Sophie", "Joanna", "Zoe", "Catherine", "Marina", "Stella",
                "Vicky", "Jenny"
        };

        thing = rand() % 6;
        sex = rand() % 2;

        namecnt = sex ? sizeof(boys)/sizeof(boys[0]) :
sizeof(girls)/sizeof(girls[0]);
        nameidx = rand() % namecnt;
        name = sex ? boys[nameidx] : girls[nameidx];

        p = buf;
        p += sprintf(p, "*** Thread %d: Oh no! ", thrid);
        p += sprintf(p, things[thing], name, sex ? "his" : "her");
        p += sprintf(p, "\n*** Why were there only %d teachers for %d
children?!\n",
                teachers, children);

        /* Output everything in a single atomic call */
        printf("%s", buf);
}

// συνάρτηση για την είσοδο παιδιού
void child_enter(struct thread_info_struct *thr)
{
        if (!thr->is_child) {
                fprintf(stderr, "Internal error: %s called for a Teacher
thread.\n",
                        __func__);
                exit(1);
        }

        fprintf(stderr, "THREAD %d: CHILD ENTER\n", thr->thrid);
```

```
        // κλειδώνουμε το κρίσιμο τμήμα
        pthread_mutex_lock(&thr->kg->lock);

        /* για να εισέλθει παιδί πρέπει να ισχύει: children<ratio*teachers, αν όχι τότε περίμενε να
εισέλθει δάσκαλος ώστε να μπορέσει να μπει το παιδί */
        while((thr->kg->vc) >= (thr->kg->vt) * (thr->kg->ratio) )
                pthread_cond_wait (&thr->kg->can_enter, &thr->kg->lock);
        ++(thr->kg->vc);
        pthread_mutex_unlock(&thr->kg->lock);    // αποδέσμευση mutex
}

// συνάρτηση για την έξοδο παιδιού
void child_exit(struct thread_info_struct *thr)
{

        if (!thr->is_child) {
                fprintf(stderr, "Internal error: %s called for a Teacher
thread.\n",
                        __func__);
                exit(1);
        }

        fprintf(stderr, "THREAD %d: CHILD EXIT\n", thr->thrid);

        // κλειδώνουμε το κρίσιμο τμήμα
        pthread_mutex_lock(&thr->kg->lock);
        // βγαίνει χωρίς συνθήκη
        --(thr->kg->vc);

        /* ενημερώνει την can_exit σε περίπτωση που θέλει να βγει κάποιος δάσκαλος */
        pthread_cond_signal (&thr->kg->can_exit);
        pthread_mutex_unlock(&thr->kg->lock);        // αποδέσμευση mutex
}

// συνάρτηση για την είσοδο δασκάλου
void teacher_enter(struct thread_info_struct *thr)
{
        if (thr->is_child) {
                fprintf(stderr, "Internal error: %s called for a Child thread.\n",
                        __func__);
                exit(1);
```

```
        }

        fprintf(stderr, "THREAD %d: TEACHER ENTER\n", thr->thrid);

        // κλειδώνουμε το κρίσιμο τμήμα
        pthread_mutex_lock(&thr->kg->lock);
       // μπαίνει χωρίς συνθήκη
        ++(thr->kg->vt);

       /* σε περίπτωση που περιμένουν παιδιά να μπουν ενημερώνει την can_enter */
        pthread_cond_signal (&thr->kg->can_enter);
        pthread_mutex_unlock(&thr->kg->lock);     // αποδέσμευση mutex
}

// συνάρτηση για την έξοδο δασκάλου
void teacher_exit(struct thread_info_struct *thr)
{
        if (thr->is_child) {
                fprintf(stderr, "Internal error: %s called for a Child thread.\n",
                        __func__);
                exit(1);
        }

        fprintf(stderr, "THREAD %d: TEACHER EXIT\n", thr->thrid);
        // κλειδώνουμε το κρίσιμο τμήμα
        pthread_mutex_lock(&thr->kg->lock);

        /* για να βγει ένας δάσκαλος  πρέπει να ισχύει: children<ratio*(teachers-1), αν όχι
τότε περίμενε να βγουν παιδιά μέχρι να πληρείτε η σχέση */
        while(((thr->kg->vt)-1)  * (thr->kg->ratio) < (thr->kg->vc))
                pthread_cond_wait (&thr->kg->can_exit,&thr->kg->lock);
        --(thr->kg->vt);

        pthread_mutex_unlock(&thr->kg->lock);       // αποδέσμευση mutex
}

/*
 * Verify the state of the kindergarten.
 */
void verify(struct thread_info_struct *thr)
{
```

```
        struct kgarten_struct *kg = thr->kg;
        int t, c, r;

        c = kg->vc;
        t = kg->vt;
        r = kg->ratio;

        fprintf(stderr, "            Thread %d: Teachers: %d, Children: %d\n",
                thr->thrid, t, c);

        if (c > t * r) {
                bad_thing(thr->thrid, c, t);
                exit(1);
        }
}

/*
 * A single thread.
 * It simulates either a teacher, or a child.
 */
void *thread_start_fn(void *arg)
{
        /* We know arg points to an instance of thread_info_struct */
        struct thread_info_struct *thr = arg;
        char *nstr;

        fprintf(stderr, "Thread %d of %d. START.\n", thr->thrid, thr->thrcnt);

        /* ανάλογα με την τιμή του is_child βλέπουμε αν το νήμα αναφέρεται σε παιδί ή δάσκαλο */
        nstr = thr->is_child ? "Child" : "Teacher";
        for (;;) {
                fprintf(stderr, "Thread %d [%s]: Entering.\n", thr->thrid, nstr);
                if (thr->is_child)
                        child_enter(thr);
                else
                        teacher_enter(thr);

                fprintf(stderr, "Thread %d [%s]: Entered.\n", thr->thrid, nstr);

                /*
                 * We're inside the critical section,
```

```
                 * just sleep for a while.
                 */
                /* usleep(rand_r(&thr->rseed) % 1000000 / (thr->is_child ? 10000 : 1)); */

                // κλειδώνουμε το κρίσιμο τμήμα
                pthread_mutex_lock(&thr->kg->mutex);
                verify(thr);
                pthread_mutex_unlock(&thr->kg->mutex);   // αποδέσμευση mutex

                usleep(rand_r(&thr->rseed) % 1000000);

                fprintf(stderr, "Thread %d [%s]: Exiting.\n", thr->thrid, nstr);
                /* CRITICAL SECTION END */

                if (thr->is_child)
                        child_exit(thr);
                else
                        teacher_exit(thr);

                fprintf(stderr, "Thread %d [%s]: Exited.\n", thr->thrid, nstr);

                /* Sleep for a while before re-entering */
                /* usleep(rand_r(&thr->rseed) % 100000 * (thr->is_child ? 100 : 1)); */
                usleep(rand_r(&thr->rseed) % 100000);

                // κλειδώνουμε το κρίσιμο τμήμα
                pthread_mutex_lock(&thr->kg->mutex);
                verify(thr);
                pthread_mutex_unlock(&thr->kg->mutex);   // αποδέσμευση mutex
        }

        fprintf(stderr, "Thread %d of %d. END.\n", thr->thrid, thr->thrcnt);

        return NULL;
}

int main(int argc, char *argv[])
{
        int i, ret, thrcnt, chldcnt, ratio;
        struct thread_info_struct *thr;
        struct kgarten_struct *kg;
```

```c
/*
 * Parse the command line
 */
if (argc != 4)
        usage(argv[0]);
if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0) {
        fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
        exit(1);
}
if (safe_atoi(argv[2], &chldcnt) < 0 || chldcnt < 0 || chldcnt > thrcnt) {
        fprintf(stderr, "`%s' is not valid for `child_threads'\n",argv[2]);
        exit(1);
}
if (safe_atoi(argv[3], &ratio) < 0 || ratio < 1) {
        fprintf(stderr, "`%s' is not valid for `c_t_ratio'\n", argv[3]);
        exit(1);
}


/*
 * Initialize kindergarten and random number generator
 */
srand(time(NULL));

kg = safe_malloc(sizeof(*kg));
kg->vt = kg->vc = 0;
kg->ratio = ratio;

ret = pthread_mutex_init(&kg->mutex, NULL);
if (ret) {
        perror_pthread(ret, "pthread_mutex_init");
        exit(1);
}

/* ... */

ret = pthread_mutex_init(&kg->lock, NULL);      //initialize mutex
if (ret) {
        perror_pthread(ret, "pthread_mutex_init");
        exit(1);
```

```
        }
        pthread_cond_init (&kg->can_enter, NULL);       //initialize cond
        pthread_cond_init(&kg->can_exit,NULL);          //initialize cond

        /*
         * Create threads
         */
        thr = safe_malloc(thrcnt * sizeof(*thr));

                for (i = 0; i < thrcnt; i++) {
                /* Initialize per-thread structure */
                thr[i].kg = kg;
                thr[i].thrid = i;
                thr[i].thrcnt = thrcnt;
                thr[i].is_child = (i < chldcnt);
                thr[i].rseed = rand();

                /* Spawn new thread */
               ret = pthread_create(&thr[i].tid, NULL, thread_start_fn, &thr[i]);
                if (ret) {
                        perror_pthread(ret, "pthread_create");
                        exit(1);
                }
        }

        /*
         * Wait for all threads to terminate
         */
        for (i = 0; i < thrcnt; i++) {
                ret = pthread_join(thr[i].tid, NULL);
                if (ret) {
                        perror_pthread(ret, "pthread_join");
                        exit(1);
                }
        }

        printf("OK.\n");

        return 0;
}
```

Παράδειγμα Εκτέλεσης:

```
oslabe15@skopelos:~/erg3$ ./kgarten 10 7 3
Thread 6 of 10. START.
Thread 6 [Child]: Entering.
THREAD 6: CHILD ENTER
Thread 7 of 10. START.
Thread 7 [Teacher]: Entering.
THREAD 7: TEACHER ENTER
Thread 7 [Teacher]: Entered.
            Thread 7: Teachers: 1, Children: 0
Thread 8 of 10. START.
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER ENTER
Thread 8 [Teacher]: Entered.
            Thread 8: Teachers: 2, Children: 0
Thread 9 of 10. START.
Thread 9 [Teacher]: Entering.
THREAD 9: TEACHER ENTER
Thread 9 [Teacher]: Entered.
            Thread 9: Teachers: 3, Children: 0
Thread 6 [Child]: Entered.
            Thread 6: Teachers: 3, Children: 1
Thread 5 of 10. START.
Thread 5 [Child]: Entering.
THREAD 5: CHILD ENTER
Thread 5 [Child]: Entered.
            Thread 5: Teachers: 3, Children: 2
Thread 4 of 10. START.
Thread 4 [Child]: Entering.
THREAD 4: CHILD ENTER
Thread 4 [Child]: Entered.
            Thread 4: Teachers: 3, Children: 3
Thread 3 of 10. START.
Thread 3 [Child]: Entering.
THREAD 3: CHILD ENTER
Thread 3 [Child]: Entered.
            Thread 3: Teachers: 3, Children: 4
Thread 2 of 10. START.
Thread 2 [Child]: Entering.
THREAD 2: CHILD ENTER
Thread 2 [Child]: Entered.
            Thread 2: Teachers: 3, Children: 5
Thread 1 of 10. START.
Thread 1 [Child]: Entering.
THREAD 1: CHILD ENTER
Thread 1 [Child]: Entered.
```

```
            Thread 1: Teachers: 3, Children: 6
Thread 0 of 10. START.
Thread 0 [Child]: Entering.
THREAD 0: CHILD ENTER
Thread 0 [Child]: Entered.
            Thread 0: Teachers: 3, Children: 7
Thread 6 [Child]: Exiting.
THREAD 6: CHILD EXIT
Thread 6 [Child]: Exited.
            Thread 6: Teachers: 3, Children: 6
Thread 6 [Child]: Entering.
THREAD 6: CHILD ENTER
Thread 6 [Child]: Entered.
            Thread 6: Teachers: 3, Children: 7
Thread 1 [Child]: Exiting.
THREAD 1: CHILD EXIT
Thread 1 [Child]: Exited.
Thread 7 [Teacher]: Exiting.
THREAD 7: TEACHER EXIT
Thread 7 [Teacher]: Exited.
            Thread 7: Teachers: 2, Children: 6
Thread 7 [Teacher]: Entering.
THREAD 7: TEACHER ENTER
Thread 7 [Teacher]: Entered.
            Thread 7: Teachers: 3, Children: 6
            Thread 1: Teachers: 3, Children: 6
Thread 1 [Child]: Entering.
THREAD 1: CHILD ENTER
Thread 1 [Child]: Entered.
            Thread 1: Teachers: 3, Children: 7
Thread 9 [Teacher]: Exiting.
THREAD 9: TEACHER EXIT
^C
```

Απαντήσεις στις ερωτήσεις:

1. Έστω ότι ένας από τους δασκάλους έχει αποφασίσει να φύγει, αλλά δεν μπορεί ακόμη να το κάνει καθώς περιμένει να μειωθεί ο αριθμός των παιδιών στο χώρο (κρίσιμο τμήμα). Τι συμβαίνει στο σχήμα συγχρονισμού σας για τα νέα παιδιά που καταφτάνουν και επιχειρούν να μπουν στο χώρο;

Εφόσον οι υπάρχοντες δάσκαλοι πληρούν την προϋπόθεση για να μπει ένα παιδί, τότε αυτό θα μπει ανεξαρτήτως αν περιμένει ένας δάσκαλος για να φύγει. Ο

δάσκαλος θα φύγει όταν έχουν φύγει αρκετά παιδιά ώστε οι υπόλοιποι δάσκαλοι να αναλογούν στα παιδιά που έχουν μείνει.

2. Υπάρχουν καταστάσεις συναγωνισμού (races) στον κώδικα του kgarten.c που επιχειρεί να επαληθεύσει την ορθότητα του σχήματος συγχρονισμού που υλοποιείτε; Αν όχι, εξηγήστε γιατί. Αν ναι, δώστε παράδειγμα μιας τέτοιας κατάστασης.

Οι καταστάσεις συναγωνισμού προκύπτουν όταν δύο νήματα εκτελούν ταυτόχρονα τα κρίσιμα σημεία και επιχειρούν να αλλάξουν δεδομένα. Στο πρόγραμμα που έχουμε φτιάξει δεν υπάρχουν καταστάσεις races καθώς σε όλα τα κρίσιμα σημεία χρησιμοποιούνται κλειδώματα mutexes.